Modeling Nested for Loops with Explicit Parallelism in Synchronous DataFlow Graphs

Alexandre Honorat, Karol Desnos, Maxime Pelcat, Jean-François Nezan

▶ **To cite this version:**

Alexandre Honorat, Karol Desnos, Maxime Pelcat, Jean-François Nezan. Modeling Nested for Loops with Explicit Parallelism in Synchronous DataFlow Graphs. Embedded Computer Systems: Architectures, Modeling, and Simulation, Jul 2019, Pythagorion, Samos Island, Greece. pp.269-280, 10.1007/978-3-030-27562-4_19 . hal-02267487v1

HAL Id: hal-02267487

https://hal.science/hal-02267487v1

Submitted on 19 Aug 2019 (v1), last revised 4 Sep 2019 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Modeling Nested `for` Loops with Explicit Parallelism in Synchronous DataFlow graphs*

Alexandre Honorat[1], Karol Desnos[1], Maxime Pelcat[1,2], and Jean-François Nezan[1]

[1] Univ Rennes, INSA Rennes, CNRS, IETR - UMR 6164, F-35000 Rennes, France
firstname.lastname@insa-rennes.fr
[2] Institut Pascal, UCA, SIGMA, CNRS - UMR 6602, Clermont-Ferrand, France

Abstract. A common problem when developing signal processing applications is to expose and exploit parallelism in order to improve both throughput and latency. Many programming paradigms and models have been introduced to serve this purpose, such as the Synchronous DataFlow (SDF) Model of Computation (MoC). SDF is used especially to model signal processing applications. However, the main difficulty when using SDF is to choose an appropriate granularity of the application representation, for example when translating imperative functions into SDF actors. In this paper, we propose a method to model the parallelism of perfectly nested `for` loops with any bounds and explicit parallelism, using SDF. This method makes it possible to easily adapt the granularity of the expressed parallelism, thanks to the introduced concept of SDF *iterators*. The usage of SDF iterators is then demonstrated on the Scale Invariant Feature Transform (SIFT) image processing application.

Keywords: SDF · parallelism

1 Introduction

Signal processing applications are generally compute intensive and constrained in terms of throughput and latency. For example, the throughput of video displays is constrained in Frame Per Second (FPS). Parallelization of such applications is the key to meet their throughput and latency requirements: when possible, data are processed simultaneously by different Processing Elements (PEs).

Parallelization of `for` loops can be achieved automatically in the code through OpenMP, or directly using threads. However, it is not possible to handle all the cases with OpenMP, as distributed memory; moreover threads require to manually add synchronizations and communications in the code. Thus, applications are usually *modeled*, in order to, first, expose their parallelism, and secondly, analyze this available parallelism and synthesize efficient schedules. A common

* This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement №732105 (project CERBERO) and from the Région Bretagne (France) under grant ARED 2017 ADAMS. We also thank Antoine Morvan and Florian Arrestier for their valuable comments.

model for signal processing applications is the Synchronous DataFlow (SDF) [10] Model of Computation (MoC), in which applications, e.g. video encoders [16], are modeled by SDF graphs. Vertices of SDF graphs encapsulate the processing code while edges model the data transfers. SDF graphs express parallelism in two ways: by the different paths in the graphs (task parallelism), and by the possible executions of the same process on different chunks of data (data parallelism). Only static applications where all communications are known in advance, and fixed, can be modeled by SDF graphs. Thus, it is possible to derive static schedules from SDF graphs. One can model single `for` loops by SDF graphs, as long as the loop can be divided in sub-parts accessing chunks of data of equal size, to respect the restriction of fixed amount of data communication. However, there is no general technique to model multiple nested `for` loops in the SDF MoC, especially when bounds of the inner loops are varying.

The contribution of this paper is the modeling, by SDF graphs, of multiple perfectly nested loops with explicit parallelism and variable bounds in their inner loops. In loops with explicit parallelism, all iterations are independent. Perfectly nested loops perform computation only in the innermost loop. This contribution is motivated by two facts. First, variable amounts of data can be modeled by the Cyclo-Static DataFlow (CSDF) [2] MoC, an extension of SDF; but previous experiments on modeling using the CSDF MoC have shown that this model is not easy to understand for designers and does not always offer a competitive benefit. This has been stated by the creators of the SDF-based language StreamIt [21], in a review of their own work [20] (see section 5.2). Another option is to use dynamic dataflow MoCs such as Kahn Process Networks (KPN) [8], but KPNs are hard to analyse and are not statically schedulable. Hence, we focus on SDF graphs instead. Second, we need to model nested loops in SDF graphs in order to finely control the granularity of the application representation. Moreover, the representation should be easily adaptable to the target architecture, especially to its number of PEs, while staying independent from the architecture.

A direct application of this contribution is the modeling of a computer vision feature detection application. Indeed, keypoints detection is performed on images at different resolutions and different blur levels. Thus, nested loops iterate over images of different sizes so the loops have variable bounds.

In this paper, we introduce the notion of SDF *iterators* modeling and optimize multiple nested loops with variable bounds. Iterators are demonstrated on a Scale Invariant Feature Transform (SIFT) keypoints detection [13] application, modeled by an SDF graph. Iterators help modeling and parallelizing SIFT detection, although some nested loops process images of variable sizes. At the same time, iterators help reducing the scheduling complexity since it is possible to adapt the number of parallel executions with regard to the number of PEs.

The paper is organized as follows. SDF graphs are presented in section 2, as well as the SIFT application that will illustrate different examples along the paper. Then the parallelization of single loops with SDF graphs is recalled in section 3. The main contribution, SDF iterators for multiple perfectly nested loops with explicit parallelism and with variable bounds, is detailed in section 4.

Results of an evaluation of iterators on SDF are presented in section 5. Related work, in section 6, is followed by a conclusion.

2 Context

Section 2.1 recalls briefly the semantics of SDF graphs, while section 2.2 presents an overview of the SIFT application that is later modeled by SDF graphs.

2.1 SDF graphs

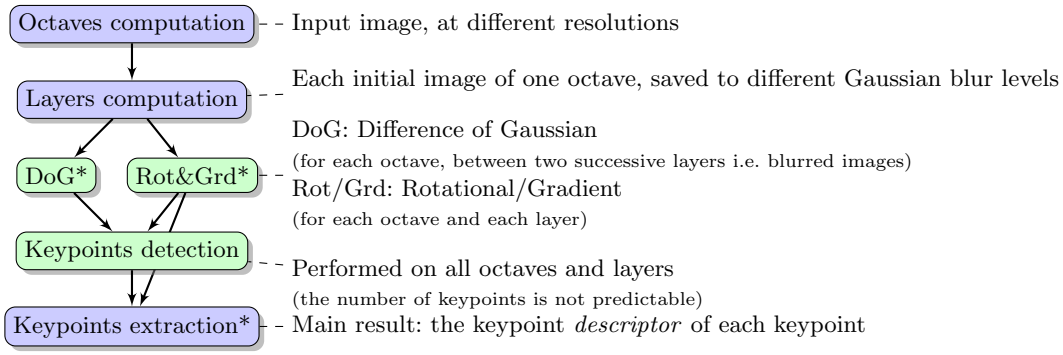
SDF graphs are directed multi-graphs composed of vertices, called *actors*, and edges, called *buffers*. Actors represent processing operations, while buffers represent the data communication between the different actors. The abstract unit of data is called *token*. Each buffer b is annotated with rates: a production rate $\text{prod}(b) \in \mathbb{N}^*$ at the source of b, and a consumption rate $\text{cons}(b) \in \mathbb{N}^*$ at the target of b. Production and consumption rates may not be equal: this is how data parallelism is expressed in SDF graphs. For example if an actor a_1 sends 6 tokens to an actor a_2 through buffer b, whereas a_2 expects 3 tokens at the other end of the buffer b, it means that a_2 will be executed twice: once on the first 3 tokens and another time on the last 3 tokens on b.

The minimal number of executions of each actor in order to leave all buffers as the same state as initially, is called a *repetition vector* and is derived from the buffers production and consumption rates. The repetition vector does not always exist, for example when rates in graph loops are not consistent and lead to a buffer overflow or underflow.

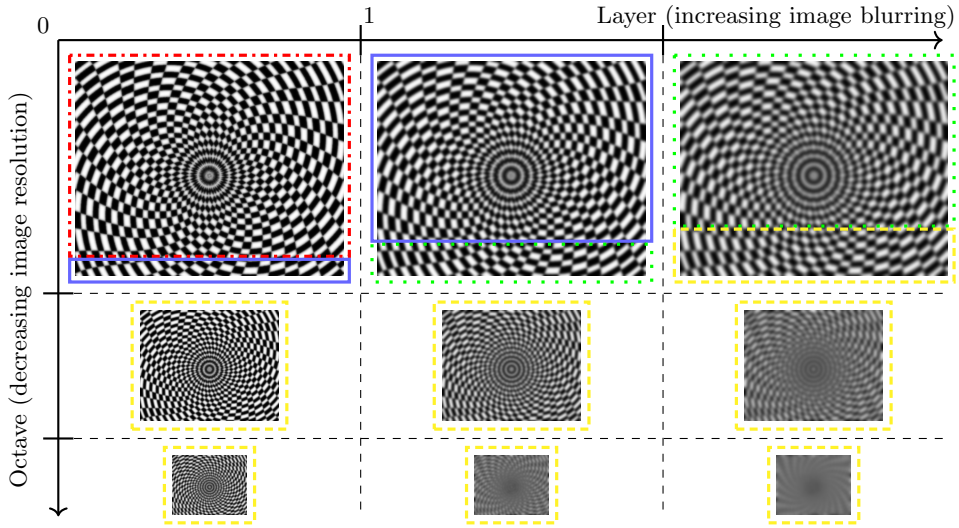
2.2 SIFT keypoints detection application

Scale Invariant Feature Transform (SIFT) computes keypoints by comparing points in the original image with the same points in blurred images obtained from the original one, and at different resolutions. Figure 1a details the main steps: first the original image is upscaled once, and downscaled several times to build the images at various resolutions. Each resolution is called an *octave*. Then, the image at each octave is blurred several times. A blur level corresponds to a *layer*. All images are stored in a 4-dimensional (4-D) array; the dimensions are, in order: octave, layer, height, width. Difference of Gaussians (DoG), gradient, and rotational metrics are computed from this 4-D array. Each metric computation produces an array of the same size, except the DoG which produces one less layer. At last, keypoints detection is performed on these three 4-D arrays. Then, the extraction step refines the computed keypoints.

Two main problems arise when modeling SIFT detection by an SDF graph. First, the number of keypoints to detect is unknown since it depends on the image content. Second, the images to process stored in the 4-D array have different sizes depending on their octave, whereas the SDF MoC imposes data transfers of fixed size. The problem on number of keypoints is easily fixed by setting a maximum.



(a) SIFT workflow: green steps* are modeled by iterators in a SDF graph.



(b) Layers and octaves in SIFT with four different region of equal processing amount.

Fig. 1: SIFT image processing application: main steps 1a and data storage 1b.

For the second problem, the naive way to model different octaves is to create a specific actor for each image size, which is not convenient because the model cannot be adapted to different numbers of octaves. Another difficulty is that the computation on the smallest image resolution, i.e. the last octave, is faster than the computation of the first octave by several orders of magnitude. Indeed, for an image of 640×800 pixel, the image is upscaled once and downscaled five times by a factor 2 on each dimension; thus the ratio of the number of pixels in the first octave over the last is $4^6 = 4096$. If several Processing Elements (PEs) are available, an important question is how to parallelize the computation equally among them. Figure 1b illustrates this problem with three layers, three octaves, and four available PEs. One option is to assign each layer to a PE, but then a PE is not used. The opposite option is to assign each octave to a PE, but

then a PE is not used, and computations are unbalanced. An example of equal distribution of the computation on the four PEs is shown in the boxes of the four colors red, blue, green and yellow (each with a specific pattern). Each color encloses one quarter of the computation. It is clear on that example that boxes do not match the image bounds.

The iterators introduced in section 4 can handle this computation partitioning while staying in the SDF model. Iterators are used to model and parallelize the green steps in the workflow in 1a, in our case according to the number of available PEs, and without duplicating any actor for each octave. Before describing the iterators in details, modeling and parallelization of single loops by SDF graphs is recalled in section 3.

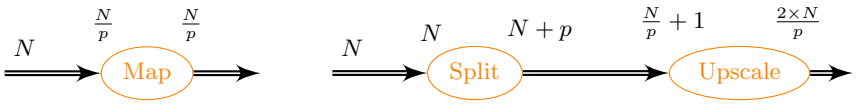
3 Modeling of single loops with explicit parallelism

This section discusses the modeling of single `for` loops using SDF graphs. `for` loops are a basic control structure of any imperative language. The code in listing 1 illustrates a simple `for` loop. It iterates over an `input` array, processes each element and stores the result in an `output` array; both arrays having the same size `N`, it represents a map operation. The parallelism is explicit: there is no dependency between the iterations of the loop, and `process` is a pure function.

```
for (int i = 0; i < N; ++i) {
  output[i] = process(input[i]);}
```

Listing 1: Simple one dimensional (1-D) `for` loop, with explicit parallelism.

Figure 2a depicts the modeling by a SDF graph of a map operation with a controllable degree of parallelism. p is the degree of expressed data parallelism: the Map actor is executed p times, on chunks of data of size $\frac{N}{p}$. If $p = N$, all data parallelism is expressed, however, it is not always useful to express all the parallelism, especially if the amount of PEs is way smaller than N. The code of the actor `Map` is almost the same as in listing 1; the only difference lies in the loop index bound that is now `N/p` instead of `N`. p must be a divisor of N.



(a) Map in SDF. (b) 1-D Upscale in SDF.

Fig. 2: Map and Upscale at a coarse-grain level.

In image processing, a common operation is to perform an upscale, increasing the resolution of the image with interpolation. This operation is more generic than map since the output array has not the same size as the input, and since several elements of the input array are accessed simultaneously to perform the interpolation. A code example is shown in listing 2 for the 1-D case.

```
for (int i = 0; i < N; ++i) {
  output[2*i] = input[i];
  if (i < N-1) {
    output[2*i+1] = interpolation(input[i], input[i+1]);
  } else {
    output[2*i+1] = input[i];}}
```

Listing 2: Simple 1-D upscale, by interpolation on the element and its successor.

The modeling of an upscale is similar to a map, but it requires extra data to apply the interpolation on the borders of the chunks of the original array. The last element of a chunk is a copy of the first element of the next chunk. These extra data can be added by a copy actor preceding the upscale actor. The SDF modeling of an upscale operation is depicted in fig. 2b, where the actor performing the interpolation is called Upscale, and the copy actor is called Split. Split is executed only once, while Upscale is executed p times. The code of the Upscale actor, in listing 3, is simpler than the original one, in listing 2, since the border case needs no more to be handled thanks to the copy performed by Split.

```
for (int i = 0; i < N/p; ++i) {
  output[2*i] = input[i];
  output[2*i+1] = interpolation(input[i], input[i+1]);}
```

Listing 3: Upscale SDF actor code.

The upscale modeling pattern presented in fig. 2b is used to model the computation of the upscale of the input image in the first step of SIFT, as shown in fig. 1a. An image has two dimensions but the data parallelism is expressed only on the height of the image, divided by the number of PEs. The same pattern is also used for the second step of SIFT: the layers computation. However, the algorithm to compute the different layers consists of two successive 1-D Gaussian blurs on lines of the image, each blur performing a transposition. The Gaussian blur applies a 1-D stencil with two neighbors. As data parallelism is expressed through the height of the image in any case, data must be reordered between the

two transpositions; this is creating an application bottleneck since this reordering is fully sequential. We now generalize the SDF modeling patterns seen in this section for single `for` loops to perfectly nested loops with explicit parallelism.

4 Modeling of nested loops with explicit parallelism

In this section, perfectly nested loops with explicit parallelism are considered. An example is given in listing 4, with three nested loops. The index bounds of the inner loops may depend on the outer loop indexes, as abstracted by the functions `f1` and `f2`, which can be any mathematical function. The parallelization of nested loops of the same form than in listing 4 is described in section 4.1, and their modeling by SDF graphs with iterators is discussed in section 4.2.

```
for (int i = 0; i < N1; ++i) {
  for (int j = 0; j < f1(i); ++j) {
    for (int k = 0; k < f2(i, j); ++k) {
      output[i][j][k] = process(input[i][j][k]);}}}
```

Listing 4: Three nested `for` loops, with explicit parallelism.

4.1 Iteration space splitting

An important property of the SDF MoC is that the rates of data exchanges are fixed. Thus, the only solution to model by SDF graphs loops as in listing 4 is to split the whole iteration space into chunks of equal sizes. These chunks do not always match the loop bounds as depicted in fig. 1b. In listing 4, the whole iteration space size $\mathcal{S}_{it}$ is $\sum_{i=0}^{N1}\left(\sum_{j=0}^{f1(i)} f2(i,j)\right)$. In this example, the iteration space size equals the total size of the array to process, it is a map operation.

The most straight forward way to cut the whole iteration space into chunks of equal size is to simulate the execution of the loop. A variable *iter* storing the total number of performed iterations is incremented instead of calling the process function. Each time the *iter* variable reaches a multiple of `chunk_size`, the loop indexes are recorded and will be used as start/stop indexes for the real execution of the loops. This algorithm is written in listing 5, with `chunk_size` being equal to any divisor of $\mathcal{S}_{it}$. The role of a SDF iterator is to send the recorded indexes to split the real execution of the loops.

Note that this simulation can be done offline: the start/stop indexes only need to be saved in order to be used during the real execution of the loop (where the process is performed). Besides, this simulation can be easily adapted to any number d of nested loops: the structure is the same as the original nested loops.

```
int iter = 0;
for (int i = 0; i < N1; ++i) {
  for (int j = 0; j < f1(i); ++j) {
    for (int k = 0; k < f2(i, j); ++k) {
      if (iter++ % chunk_size == 0) {
        record(i,j,k); }}}}
```

Listing 5: Iteration space simulator for three perfectly nested `for` loops with explicit parallelism. The recorded indexes will be stored in the SDF iterator.

4.2 SDF iterators

An SDF iterator actor provides the start and stop indexes for each execution of the process actor modeling the nested loops. Thus, if the nested loops iteration space $\mathcal{S}_{it}$ is divided into p chunks, the iterator is executed once and the processing actor is executed p times. The code for the processing actor is similar to its original version in listing 4, the only difference concerns the indexes that are set by the iterator output. The modeling of the perfectly nested loops in SDF graphs is depicted in fig. 3. $\mathcal{S}_{it}$ elements are sent to the Process actor, which is processing them by chunks of size $\frac{\mathcal{S}_{it}}{p}$. For each execution of Process, the iterator produces $2 \times d$ indexes: one start index and one stop for each loop of the d nested loops. These indexes correspond to the one recorded during the loop simulation presented in listing 5, considering that the stop indexes of one execution of the process are the start indexes of its next execution. Hence, the p executions of Process can be performed in parallel. Note that the map and upscale patterns described in fig. 2b can also be applied to this general case of nested loops.

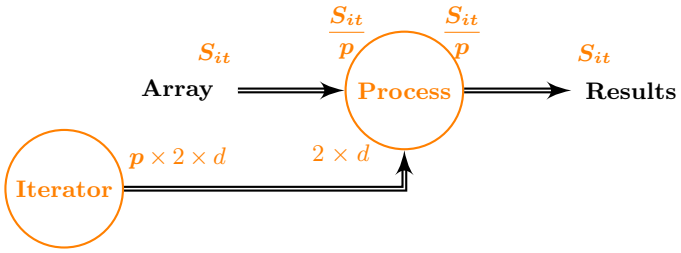


Fig. 3: Modeling of iterators in SDF.

Finally, iterators empower the designer to control data parallelism in SDF graphs for any perfectly nested loops with explicit parallelism. Iterators induce two drawbacks which can be overcome. The first drawback occurs when tagging an actor a, parallelized with an iterator, with a measured execution time C_a.

Indeed, the execution time is then not only dependant on the actor code, but also on the chunk size. This drawback is easily overcome using symbolic expressions of the execution time: $C_a(p) = \frac{C_a(1)}{p}$; that is, when a is parallelized over p chunks, the execution time of each execution of a is equal to its sequential execution time divided by the number of chunks. The second drawback is the restriction to nested loops with explicit parallelism. However, this restriction can be removed in some cases thanks to prior source-to-source code transformations.

5 Evaluation

SDF iterators are now used to model the SIFT keypoints detection application. In SIFT detection, images from different resolutions and blur levels are processed and this implies to iterate over a 4-D array with four perfectly nested loops. The two innermost loops iterating over the height and width of images have exponential bounds depending on the top loop index. The top loop iterates over the octaves and the second loop iterates over the layers. If octaves are indexed by the variable i, the image height (respectively, width) to be processed is the biggest resolution height (resp. width) divided by 2^i. SDF iterators are used to model such loops, expressing a degree of parallelism $p \in \{1, 2, 4, 5, 10, 20\}$ (this set contains the common divisors of the sizes of all the 4-D arrays).

The SIFT detection code is a slightly modified version of the ezSIFT[3] implementation. The SDF model of SIFT is built under the PREESM [17] framework, which also performs static scheduling and generates the static parallelized code. Another parallel version of SIFT has been implemented using OpenMP, mainly with `parallel for` pragma above the loops iterating over the height of the images. The keypoint detection step requires a critical section to add the detected keypoints into a shared list.

The execution times of the PREESM version and the OpenMP version of SIFT have been reported in table 1. All experiments used an Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz processor and the GCC compiler version 5.4.0. Both PREESM and OpenMP execution times are similar, the best speedup (in bold) is achieved alternatively by PREESM and OpenMP.

#Cores	#Tasks	PREESM Time (Speedup)	OpenMP Time (Speedup)
1	190	669 (0.96x)	645 (**ref.**)
2	293	412 (1.56x)	406 (**1.59x**)
4	355	277 (**2.33x**)	281 (2.29x)
5	386	263 (2.45x)	255 (**2.53x**)
10	541	171 (**3.77x**)	182 (3.54x)

Table 1: Number of scheduled tasks, execution times in ms, and speedup for different number of cores. Execution time is an average on 200 runs.

[3] See code on: `https://sourceforge.net/projects/ezsift/`

The number of scheduled tasks when unfolding the SDF graph of SIFT is also reported in table 1. This number is the sum of the repetition vector, i.e. the sum of the minimal number of executions of each actor. As PREESM supports parameterized SDF graphs [5], first introduced in [1], the expressed degree of parallelism p is set according to the number of cores. The number of tasks is not multiplied by a factor equal to the number of cores since not all steps of SIFT are parallelized through iterators, as shown in fig. 1a.

This evaluation shows that it is possible to model and parallelize an application having perfectly nested `for` loops with variable index bounds thanks to SDF iterators. We achieve competitive performances against OpenMP, that we could even improve by adding delays in the SDF graph (delays create pipelining). Finally, we are able to control the expressed degree of parallelism, although restricted to be a divisor of the size of the considered iteration space.

6 Related work

The modeling of nested `for` loops in SDF graphs thanks to iterators is related to two main aspects: specialized dataflow languages, especially for image processing applications where at least two dimensions are considered, and dataflow graph clustering, since iterators impact on the degree of data parallelism.

6.1 On specialized dataflow languages

The most relevant dataflow MoC for image processing is the Multidimensional Synchronous DataFlow (MDSDF) MoC [14], expressing data parallelism across several dimensions. However this MoC does not solve the problem of variable image sizes such as in the SIFT octaves: it needs one specialized actor per size. Moreover, only a few tools [9], such as ArrayOL [3], support the MDSDF MoC.

The Brook stream language [12] supports a subset of MDSDF graphs, expressed directly as `C++` code. Brook only handles kernels with affine bounds, and thus cannot be used for variable image sizes inducing exponential bounds. More generally, the same problem arises for all models relying on polyhedral analysis [6], as the Polyhedral Process Network (PPN) [22], a parameterized extension of it [26], or the OpenStream extension of OpenMP [4]: they are dedicated to loops with affine bounds only. Extensive analyses and graph and code transformations allow to model any kind of loops in PPNs [15] but then do not offer control of the degree of parallelism. Dynamic dataflow languages, such as the one supported by Orcc [23], offer more flexibility on the application representation, however it is not possible to derive static schedules from such languages when their semantics are fully exploited.

6.2 On the clustering of dataflow graphs

Clustering is usually performed on graphs expressing more parallelism than available on the target architecture; clustering simplifies scheduling without increas-

ing the application run time [19]. The standard way to reach a coarse representation is to limit the unfolding of the SDF graph into a precedence task graph. The unfolding is limited by merging different actors, or several executions of the same actor. This operation artificially reduces the repetition vector size, or decreases the values held by the repetition vector. This method has been employed for SDF graphs under real-time constraints [25], and also under data-driven scheduling of Partial Expansion Graphs (PEG) [24]. The StreamIT benchmark has also been successfully transformed into coarse SDF graphs for the RAW architecture [7], by an unfolding technique using actor fusion and fission.

Another method is to completely unfold the SDF graph, and only then, to apply clustering algorithms; however, this significantly increases the scheduling complexity. Clustering algorithms exist for precedence graphs, including compiler intermediate representations [11]. Hierarchical SDF graphs have also been used to model nested loops [18], one per hierarchy, but they require an analysis of the iteration space. Both methods do not offer control on the degree of parallelism.

Finally, all the aforementioned clustering methods rely on algorithms that analyze the graph and create a coarse or hierarchical version of it, while the proposed iterators only require to replace the `for` loop index bounds by the iterator output. Only one iterator per iteration space is needed.

7 Conclusion

This paper has demonstrated that it is possible to model a subclass of nested loops with variable index bounds by an SDF graph, and to control the degree of expressed parallelism. Thus, we can add to the observations of the StreamIt creators that the CSDF model is not only complicated to use, but is also not always compulsory: the SIFT application has been parallelized efficiently thanks to the SDF model and iterators. A future extension of this work is to automatically generate iterator code from a code analysis.

References

1. Bhattacharya, B., Bhattacharyya, S.S.: Parameterized dataflow modeling for dsp systems. IEEE TRANSACTIONS ON SIGNAL PROCESSING **49** (2001)
2. Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J.: Cycle-static dataflow. Trans. Sig. Proc. **44**(2) (1996)
3. Boulet, P.: Array-OL Revisited, Multidimensional Intensive Signal Processing Specification. Research Report RR-6113, INRIA (2007)
4. Cohen, A., Darte, A., Feautrier, P.: Static Analysis of OpenStream Programs. Research Report RR-8764, CNRS ; Inria ; ENS Lyon (2016)
5. Desnos, K., Pelcat, M., Nezan, J.F., Bhattacharyya, S.S., Aridhi, S.: PiMM: Parameterized and Interfaced dataflow Meta-Model for MPSoCs runtime reconfiguration. In: 13th International Conference on Embedded Computer Systems: Architecture, Modeling and Simulation (SAMOS XIII). Samos, Greece (2013)
6. Feautrier, P.: Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. International Journal of Parallel Programming **21**(6) (1992)

7. Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. SIGPLAN Not. **41**(11) (2006)
8. Kahn, G.: The semantics of simple language for parallel programming. In: IFIP Congress (1974)
9. Keinert, J., Deprettere, E.F.: Multidimensional Dataflow Graphs. Springer New York, New York, NY (2013)
10. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proceedings of the IEEE **75**(9) (1987)
11. Li, F., Pop, A., Cohen, A.: Automatic Extraction of Coarse-Grained Data-Flow Threads from Imperative Programs. IEEE Micro **32**(4) (2012)
12. Liao, S., Du, Z., Wu, G., Lueh, G.Y.: Data and computation transformations for brook streaming applications on multiprocessors. In: International Symposium on Code Generation and Optimization (CGO'06) (2006)
13. Lowe, D.G.: Distinctive image features from scale-invariant keypoints. International Journal of Computer Vision **60**(2) (2004)
14. Murthy, P.K., Lee, E.A.: Multidimensional synchronous dataflow. IEEE Transactions on Signal Processing **50**(8) (2002)
15. Nadezhkin, D., Nikolov, H., Stefanov, T.: Automated generation of polyhedral process networks from affine nested-loop programs with dynamic loop bounds. ACM Trans. Embed. Comput. Syst. **13**(1s) (2013)
16. Oh, H., Ha, S.: Fractional rate dataflow model and efficient code synthesis for multimedia applications. SIGPLAN Not. **37**(7) (2002)
17. Pelcat, M., Desnos, K., Heulot, J., Guy, C., Nezan, J.F., Aridhi, S.: Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming. In: 6th Embedded Design Education and Research Conference (2014)
18. Piat, J., Bhattacharyya, S.S., Raulet, M.: Loop transformations for interface-based hierarchies IN SDF graphs. In: 21st IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP). Rennes, France (2010)
19. Pino, J.L., Bhattacharyya, S.S., Lee, E.A.: A hierarchical multiprocessor scheduling system for dsp applications. In: Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers. vol. 1 (1995)
20. Thies, W., Amarasinghe, S.: An empirical characterization of stream programs and its implications for language and compiler design. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques. PACT '10, ACM, New York, NY, USA (2010)
21. Thies, W., Karczmarek, M., Amarasinghe, S.P.: Streamit: A language for streaming applications. In: Proceedings of the 11th International Conference on Compiler Construction. CC '02, Springer-Verlag, London, UK, UK (2002)
22. Verdoolaege, S.: Polyhedral Process Networks. Springer US, Boston, MA (2010)
23. Yviquel, H., Lorence, A., Jerbi, K., Cocherel, G., Sanchez, A., Raulet, M.: Orcc: Multimedia development made easy. In: Proceedings of the 21st ACM International Conference on Multimedia. MM '13, ACM (2013)
24. Zaki, G.F., Plishker, W., Bhattacharyya, S.S., Fruth, F.: Implementation, scheduling, and adaptation of partial expansion graphs on multicore platforms. Journal of Signal Processing Systems **87**(1) (2017)
25. Zhai, J.T., Bamakhrama, M.A., Stefanov, T.: Exploiting just-enough parallelism when mapping streaming applications in hard real-time systems. In: 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC) (2013)
26. Zhai, J.T., Nikolov, H., Stefanov, T.: Modeling adaptive streaming applications with parameterized polyhedral process networks. In: 2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC) (2011)

Addendum

This addendum is not part of the original accepted paper: it has been added by A. Honorat and has not been reviewed. It focus on two points:

- standard usage of SDF iterators: when it is useful;
- actor code transformation with start and stop indexes of loops.

Also, one related work has been published at the same time of this paper. Leben et al. applied polyhedral techniques to multi-dimensional streams[4] , not using dataflow models. I disagree with their statement in the introduction of their paper: "*Multi-dimensional streams can be represented in such models* [Synchronous DataFlow (SDF)]*, albeit at the cost of abstraction, which makes this less natural for the programmer and restricts potential transformations in the compiler.*". Regarding the first assumption, it depends on the qualification of the programmer: if the programmer is used to the LabVIEW software for example, the SDF model will be perfectly natural for him/her. Regarding the second assumption, it is actually the opposite since we demonstrate in this paper that we can model loops with non affine bounds.

When SDF iterators are needed?

Even when modeling perfectly nested loop with explicit parallelism, SDF iterators are not always needed. It is required to use SDF iterators only when the loop indexes are used in the processing part. For example, loop indexes enable to knowing the image bounds in the 4-D arrays of the SIFT application, and thus the processing is able to call specific code on borders of the image. The indexes are needed in SIFT also to access to the whole array (see the next subsection for an example). The implementation of SIFT, modeled with SDF iterators, is available on `https://github.com/preesm/preesm-apps`, in the `SIFT` folder.

On the contrary, SDF iterators are not strictly required for map/reduce operations, even on multidimensional data, since map/reduce operations do not need to be aware of the current position in the input array. Indeed, in this case, only the size of the iteration space is needed. However, iterator may still be used, for example in order to handle padding data if the size of the iteration space is not a multiple of the number of processors. The map operation in the SDF MoC is depicted in fig. 2a in the paper, while the reduce operation is depicted below in fig. 4. The reduce operation requires to create $\log_c N$ reduce actors in the corresponding SDF graph in order to control the parallelism level; each reduce actor consumes $\frac{N}{C}$ elements and produces 1. The repetition vector of these reduce actors is (in the same order as in the graph): $[\frac{N}{C}, \frac{N}{C^2}, \frac{N}{C^3}, ..., 1]$. It requires that the total number of elements N is a power of the chunk size C.

[4] Polyhedral Compilation for Multi-dimensional Stream Processing, Jakob Leben et al., ACM Transactions on Architecture and Code Optimization, Vol. 16, No. 3, Article 27. Publication date: July 2019.

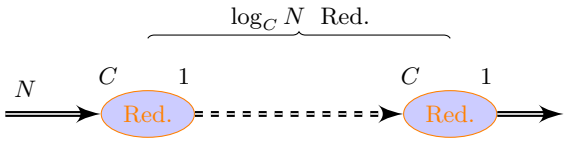


Fig. 4: Reduce of N elements in SDF, with chunk size $C > 1$.

About the restriction to perfectly nested loops with explicit parallelism, note as well that the bounds of the loops cannot depend on the input data. If loops are not perfectly nested, it is still possible to use SDF iterators; however, then the computation times of the actor firings may be unbalanced.

How the processing code must be modified?

Let's consider the code in listing 6, that has to be modeled in an SDF graph and parallelized. This piece of code contains two nested non-affine loops, iterating over a 2-D array stored in a row-major fashion (outer loop dimension first). The array contains 14 elements in total and will be split in two chunks, to be parallelized on two PEs.

```
float* array = float[1+2*2+3*3]; // = 14
// fill array with file data
for (int i = 0; i < 3; i ++) {
  int bound = (i+1)*(i+1);
  for (int j = 0; j < bound; j ++) {
    processing1cell(array, i, j);
  }
}
```

Listing 6: Non affine 2-D `for` loop, with explicit parallelism.

When adding the iterator in the SDF graph, the code of the actor to parallelize has to be adapted to its new inputs: the start and stop indexes of the loops. The modified code is presented in listing 7. In the inner loops, the bounds cannot be used directly: they are used only if the upper loop is also using its start (respectively stop) index.

Note that, in this example, not the full array is considered for the processing. The array is split in two chunks, and thus the SDF processing actor is fired twice by the SDF application execution framework, PREESM in our case. PREESM will feed each chunk with the correct data. The order of firings follows the order of the data: the second firing gets the second chunk (elements 7 to 13). It is also

```
float* array = ; //provided by input
for (int i = start_i; i < stop_i; i ++) {
  int begin_j = (i == start_i) ? start_j : 0;
  int end_j = (i+1 == stop_i) ? stop_j : (i+1)*(i+1);
  for (int j = begin_j; j < end_j; j ++) {
    processing1cell(array, i, j);
  }
}
```

Listing 7: Non affine 2-D `for` loop SDF actor code.

possible to use the full array, without extra memory consumption. This situation occurs in SIFT where the detection and extraction of the keypoints depends on pixels at different octaves, and thus depends on data spread everywhere in the 4-D array. To do so, a so-called `Broadcast` actor is added to the SDF graph, and provides multiple *virtual* copies of the 4-D input array: there are as much copies as there are firings of the processing actor. These copies are virtual since they refer to the same physical memory, by using the memory scripts introduced by Karol Desnos et al. [5]. A graph extract corresponding to this solution is depicted in fig. 5. In this solution, the `Process` actor is still fired p times, and still processes only $\frac{S_{it}}{p}$ elements of the array, however it sees the whole array.

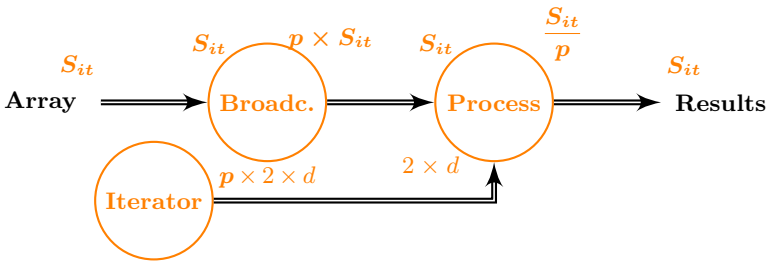


Fig. 5: Modeling of iterators in SDF with broadcast actor.

[5] On Memory Reuse Between Inputs and Outputs of Dataflow Actors, Karol Desnos et al., ACM Transactions on Embedded Computing Systems, Vol. 15, No. 2, Article 30, Publication date: February 2016.